Rendering of Spherical Light Fields

Insung Ihm, Sanghoon Park, Rae Kyoung Lee
Department of Computer Science, Sogang University
1 Sinsu-dong Mapo-gu Seoul, 121-742, Korea
{ihm,hun,lrk}@graphlab.sogang.ac.kr

Abstract

A plenoptic function is a parameterized function describing the flow of light in space, and has served as a key idea in building some of the recent image-based rendering systems. This paper presents a new representation scheme, called a spherical light field, of the plenoptic function, that is based on spheres. While methods using spherical coordinates are thought to require substantially more computation than those using planar or cylindrical coordinates, we show that spheres can also be used efficiently in representing and resampling the flow of light. Our image-based rendering algorithm is different from the previous systems, the light field and lumigraph, in that it is an "object-space" algorithm that can be easily embedded into the traditional polygonal rendering system. Our method is easily accelerated by 3D graphics boards that support the primitive functionality, such as viewing and smooth shading. In addition, we introduce an encoding scheme based on wavelets for compression of the huge data resulting from sampling of the spherical light field. The proposed technique can be easily adapted to compress the light field and lumigraph data, and offers as high compression ratios as the previous methods. Furthermore, it naturally creates a multi-resolutional representation of the light flow that can be exploited effectively in the future applications. We show how to access the compressed data efficiently using a modified significance map and an incremental decoding technique, and report experimental results on several test data sets.

1. Introduction

One of the most important goals of computer graphics is to generate realistic images from complex scenes. Several approaches to image synthesis have been developed including polygonal rendering, ray tracing, and radiosity. Application of these traditional methods involves two complicated tasks: geometric representation of three dimensional (3D) scenes and the physical description of lighting attributes of the scenes. Considerable work has been undertaken to enhance the realism of generated pictures and to reduce the computational complexities of algorithms. Despite rapid advances in modeling and rendering, creating realistic pictures of virtual environments requires great computational expense and the results are still far from real-time computation.

A different approach, called image-based rendering, has been developed recently. Unlike traditional geometry-based rendering, image-based rendering techniques generate realistic pictures from a set of pre-acquired images. Usually, the source of the pre-computed images is built from digitized photographs or from synthesized images. Using photographs from the real world makes it possible to skip most of the traditionally laborious modeling and rendering processes. The pre-processed imagery allows the implementation of rendering complex scenes with a constant amount of computation per frame, most likely at real time or interactive rates. This is so even when the source images are synthesized from the virtual world, in which case they may still go through the modeling and rendering processes.

The idea of image-based rendering has long been applied in texture mapping and environment mapping [5, 4, 13]. In Chen [6], multiple environment maps are created from cylindrical panoramic images at discrete points, and they are used to compose images seen from locations with continuously changing viewing directions. Image morphing has also been a popular technique for image-based rendering [2, 20]. In Chen et al. [7], the authors made use of the view interpolation approach to synthesize 3D scenes where regularly spaced synthetic images with their computed depth maps are smoothly interpolated as a camera moves. In Debevec et al. [9], a hybrid approach combining both geometry-based and image-based methods was presented for modeling and rendering architectural scenes from a set of still photographs.

McMillan et al. [17] presented an image-based rendering system, based on sampling, reconstructing, and resampling the flow of light. This, they called a *plenoptic function*, where a cylindrical projection was used as the plenoptic sample representation. The plenoptic function was defined as the pencil of rays visible from any point in space, at any time, over any range of wavelengths [1]. When time is fixed, the relationship is a function of five variables that describe the flow of light at a point (x, y, z) in a given direction (θ, ϕ). The 5D plenoptic function can be simplified to 4D when views of an object need to be generated from outside its convex hull. This 4D space may be considered a function of the space of oriented lines.

Two similar parameterizations of this 4D function have been presented recently. Levoy and Hanrahan [15] parameterized rays by their intersection with two parallel planes. Two pairs (s, t) and (u, v) of parameters on the planes defined an oriented ray, and they restricted the ranges of the parameters so that points on each plane lay within convex quadrilaterals. This parallelepiped of the pencils of rays was called a *light slab*. The 4D function, called a *light field*, was created using an appropriate number of such light slabs. In [12], Gortler et al. chose a cube to bound an object, and for each face, parameterized the rays in a similar way to create the 4D function, *Lumigraph*.

Our work extends previous research based on the concept of the plenoptic function. We present a new parame-

terization and representation of light flow based on spheres, called *spherical light field*. While methods using spherical coordinates are thought to require substantially more computation than those using planar or cylindrical coordinates, we show that spheres can also be used efficiently in representing and resampling the flow of light. The rendering algorithms used in previous studies [15, 12] are "image-space" algorithms in that they have the same basic structure as ray casting. Our rendering algorithm is different in that it is an "object-space" algorithm that can be embedded easily into the traditional polygonal rendering system. It is accelerated easily by 3D graphics boards that support the primitive functionality such as viewing and smooth shading, and allows a simple-to-implement hybrid rendering in which complex regions are rendered from sampled spherical light fields, while less complex regions are rendered from simple polygonal models.

Like the other approaches quoted, our image-based rendering system must handle a huge amount of data. While the successful encoding techniques of previous approaches can be incorporated into our rendering system for compression, we also introduce another possible compression scheme based on wavelets. As well as the proposed technique, that can be easily adapted to compressed the light field and lumigraph data, offers as high compression ratios as the previous methods, it naturally creates a multi-resolutional representation of the light flow, that can be exploited effectively in the future applications. We show how to access the compressed data efficiently using a modified significance map and an incremental decoding technique, and report experimental results on several test data sets.

2. Spherical Light Field Rendering

2.1. Representation of Spherical Light Fields

In this section, we propose a new parametrization of the oriented rays in 3D space. Our approach differs from previous works in that a sphere is used as the convex hull of a bounded object (See Figure 1.). Each point on the surface of the sphere is parameterized by two variables (θ_p, ϕ_p). Then, an oriented ray in the space is determined by associating a direction (θ_d, ϕ_d) with each point on the sphere. This results in a different 4D parametrization of the plenoptic function $C \equiv (R, G, B) = Ray(\theta_p, \phi_p, \theta_d, \phi_d)$, which we call a *spherical light field*.

The parametrization can be viewed as a collection of small *directional* spheres clinging to a large *positional* sphere (Figure 2). Consider the positional sphere that is a unit sphere, centered at the origin, bounding objects in the scene. Points $p = (x, y, z)$ on the sphere can be parameterized by two variables θ_p and ϕ_p ($0 \leq \theta < 2\pi$ and $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$) that represent the longitude and latitude, respectively. The direction at p is also parameterized by two variables (θ_d, ϕ_d). A natural choice of the coordinate system for directions (θ_d, ϕ_d) is the spherical frame field where the z axis is normal to the surface at p, and the x and y axes are in the directions to the parallel and meridian passing through p, respectively (Figure 2(a)) [18]. The coordinate system is natural in the sense that it is appropriate for representing the upper hemisphere that is usually enough to describe the flow of light. In our framework, however, we choose a different frame, whose origin is at p, and the three axes are parallel to those for the positional sphere. This coordinate system has the advantage that a given direction is

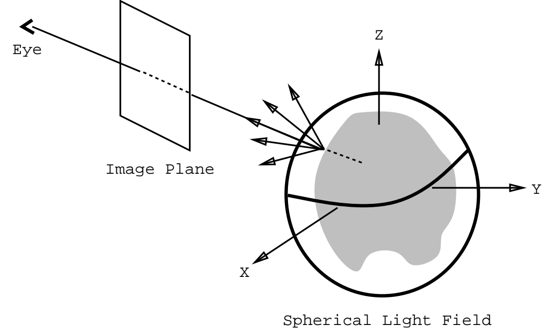


Figure 1. The Representation of a 4D Spherical Light Field

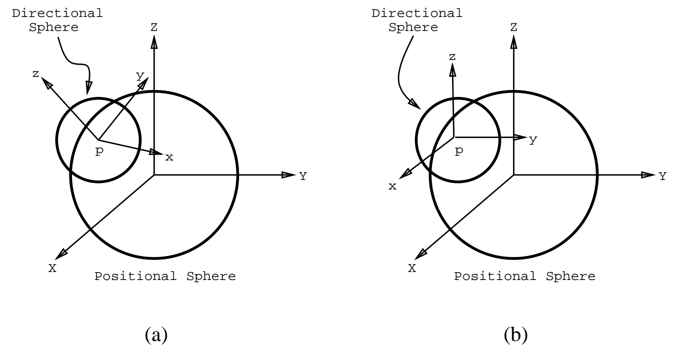


Figure 2. Two Coordinate Systems for 4D Parametrization

described by the same parameter values for all the directional spheres. This coordinate system results in efficient computation, especially when a parallel projection is used.

The use of spheres provides a symmetric representation of the complete flow of light, which makes it easy to handle arbitrary viewpoints and directions without exceptions. While spheres are generally less efficient than planes or boxes from a computational viewpoint, our rendering method is designed to access information, stored in a sphere, as efficiently as possible.

2.2. Discretization of Spherical Light Fields

The spherical light field has been defined as a function in a continuous, four dimensional space. In practice, to be used in a computational framework, the functional space must be discretized or be sampled. Mathematically speaking, the function Ray can be expressed as a combination of two functions f_p and f_d: $Ray(\theta_p, \phi_p, \theta_d, \phi_d) = f_d(\theta_d, \phi_d) = (f_p(\theta_p, \phi_p))(\theta_d, \phi_d)$, where $f_p : V \longrightarrow (V \longrightarrow C), f_d : V \longrightarrow C$, and $V = \{(\theta, \phi) | 0 \leq \theta < 2\pi, -\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}\}$. That is, f_p is a function defined on a sphere whose value is a function f_d, in turn defined on another sphere. Hence, the task of sampling the 4D spherical light fields may be reduced to the problem of the finite approximations of two spheres.

In our framework, we start from the octahedron, where each triangular face corresponds to the eight regular patches on the sphere, each corresponding to the octants of the xyz-axes. Figure 3 illustrates how each triangular face is subdivided recursively into four finer triangles. Three new vertices are selected in the centers of the circular arcs that connect the vertices of the triangles. In contrast to geodesic construction, in which the great circles of the sphere are used, we use the circles that are *orthogonal* to the axes of the coordinate system. With our tessellation scheme, it is cheaper than the geodesic tessellation to locate a triangle that contains a given direction because only simple comparisons with x, y, and z coordinates are necessary.

As mentioned previously, we need to tessellate two different spheres, the positional sphere and the directional sphere. For the positional sphere, we assume that the function f_p is discretized so that it is defined at the vertices of the approximating polyhedron, and the polygonized sphere is stored in a conventional triangular mesh form. The approximating polyhedron for the positional sphere can be any polyhedron made of an arbitrary number of vertices. One the other hand, the function f_d on the directional sphere is assumed to have values at the barycentric centers of the triangular faces, generated from recursive subdivision of the base triangles of an octahedron. Since the value of f_d is an RGB (or RGBA) color, we can imagine a flat-shaded polyhedron where the color of each triangle is the value assigned to the center of the triangle. Figure 5(a) displays an example of flat-shaded directional sphere with level-5 discretization.

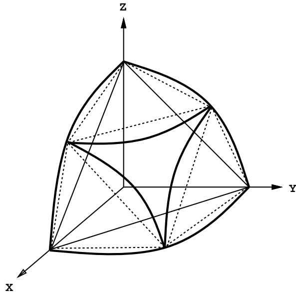
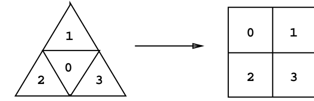


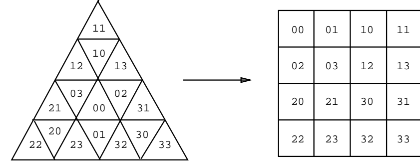
Figure 3. Recursive Subdivision of Base Triangles

The discretized spherical data for the directional spheres are reordered into a two dimensional array for effective storage and manipulation as illustrated in Figure 4. When a triangular patch is subdivided into four sub-patches, they are labeled, and are reordered into an array as in Figure 4(a). This labeling continues as the triangular paths are recursively subdivided, and each triangle of the final polyhedron is named by concatenating the labels corresponding to the region. Figure 4(b) shows an example of the level 2 subdivision and its array representation. Notice that a quadtree is implicitly constructed for each base triangle of the octahedron corresponding to one-eighth of the directional sphere. In this representation, we view the sphere as a tree that is described as follows:

- The root has two children corresponding to the upper and lower hemispheres.

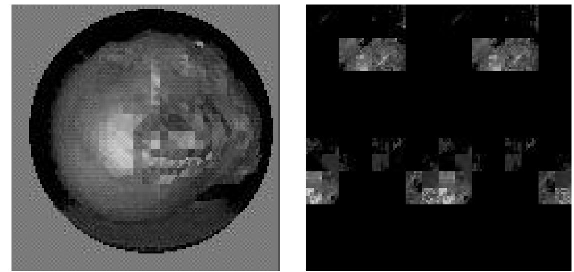


(a) Labeling order



(b) Level two subdivision

Figure 4. Reordering of Spherical Data



(a) A Flat-Shaded Directional Sphere

(b) Two Reordered Spheres

Figure 5. The Directional Spheres and Their Reordered Arrays

- The two children have four children corresponding to the four base triangles.
- The children are the roots of the quadtrees that represent the base triangles.

When the base triangle is subdivided up to level 5, $4^5 = 1024$ triangles are generated, and the data associated with the centers are stored in a $2^5 \times 2^5$ array. Each hemisphere is then put into a $2^6 \times 2^6$ array, and a whole discretized directional sphere is represented by a $2^7 \times 2^6 = 128 \times 64$ array. Figure 5(b) shows a 128×128 image that has been reordered from two spheres, where the left half is from the sphere in Figure 5(a). When the base triangle is discretized up to level 4, a directional sphere is represented by a 64×32 array.

2.3. Polygonal Rendering of Spherical Light Fields

So far, we have discussed the representation of discretized spherical light fields that are serialized into two composite spheres. We now describe how an image is displayed from a given spherical light field. Levoy et al. [15] and Gortler et al. [12] produced an image using algorithms

that have the same main for-loops as a ray tracer. For each ray generated pixel-by-pixel, the corresponding (s, t, u, v) coordinates are computed, then, a resampling process is carried out to compute the color along the ray. These methods may be classified as image-space algorithms. On the other hand, our approach is an "object-space" algorithm that is easily embedded into a polygonal rendering system.

Our viewer generates an image by rendering the triangles of the polyhedron that discretize the positional sphere. The rendering process is a smooth-shading of a polyhedron. Given a camera position and a projection type, the back-faced triangles of the sphere are culled first. For every front-faced triangle, a color is associated to each of its vertices. To do this, the projection direction for the vertex, parameterized by (θ_p, ϕ_p), is decided, and it becomes a point (θ_d, ϕ_d) on the directional sphere corresponding to the vertex. The color that is assigned to the vertex is the functional value $Ray(\theta_p, \phi_p, \theta_d, \phi_d)$ of the spherical light field. Then the triangle is projected onto an image plane, and the colors at the vertices are bilinearly interpolated at the pixels it covers.

Most of the simple rendering process can be implemented easily in a conventional polygonal rendering system such as OpenGL. The shading part is programmed in just a few OpenGL commands. First, the smooth shading model is selected by the glShadeModel () function with the mode parameter GL_SMOOTH. Then the triangles are drawn by the glVertex() functions with the colors assigned to their vertices by the glColor () functions. Back-face culling and viewing are routine in polygon-based rendering. Graphics boards that offers primitive functionality such as viewing, culling, and smooth shading, are available today at moderate prices. Hardware acceleration of spherical light field rendering is hence quite affordable even on low-end workstations or PCs.

The only major part that must be computed in software is that of extracting the color of a vertex (θ_p, ϕ_p) from the directional sphere $f_d = f_p(\theta_p, \phi_p)$. When the projection direction is (θ_d, ϕ_d), the color becomes $f_d(\theta_d, \phi_d)$. Since the directional sphere is now defined on the discretized space, accessing the function involves a resampling process. We find a simple nearest-neighbor filtering technique adequate. Filtering based on linear interpolation from the nearest samples on the directional sphere would result in few aliasing artifacts. However, interpolation-based filtering is expensive, especially when real-time rendering is desirable. As stated above, the directional sphere is subdivided into eight patches, each corresponding to an octant of the xyz-axes. Then each patch is recursively refined into four subpatches. When recursion is repeated up to level 5, we obtain $8 \times 4^5 = 8192$ directions on the discrete sphere which are enough for nearest-neighbor resampling in most cases. When the maximum level is 4, there are $8 \times 4^4 = 2048$ directions. In the case of this fewer number of directions, using nearest-neighbor filtering may cause some local distortion in a displayed image with perspective projections, because each projector is modified to its nearest one among the 2048 directions. Such a problem does not occur, however, in parallel projections because all the points on the positional sphere take on the same (nearest) direction for the directional sphere.

3. Wavelet-Based Compression

Like the light field and lumigraph, a huge amount of storage is necessary to represent a nontrivial spherical light field. For example, we used approximately 64K points for the positional sphere when an octahedron is used as a base polyhedron. When each of the eight quadtrees have level 5 for the directional sphere and 24 bits are used for a color, $64 \times 2^{10} \times 8 \times 4^5 \times 3 = 1.5$GB of storage is required. If we use quadtrees with a maximum level of 4, the storage required is 384MB.

To make implementation of spherical light fields practical, the data must be compressed. In [15], the authors describe a compression system consisting of fixed-rate vector quantization followed by entropy coding. To quantize a light field, they first partition the source data into a set of sample vectors, and construct a codebook of reproduction vectors called codewords that best approximate the samples. Then the sample vectors in the source are replaced by indices of the closest approximating codewords from the codebook. The codebook and indices are further compressed by gzip which is an implementation of Lempel-Ziv coding. They report an overall compression ratio of a factor of more than $100 : 1$, while up to $24 : 1$ compression is possible using the vector compression that decides the actual amount of run time memory. In [12], the authors guess that a $200 : 1$ compression ratio could be achievable with almost no degradation.

In this section, we propose another compression technique, designed for the spherical light field, which can be easily adapted to the light field and lumigraph. Notice that these data contain substantial coherence, hence a good compression technique must remove redundancy well. Furthermore, the compression technique must provide a low-cost random access to compressed data. Recall how the rendering process tries to access the pixels of directional spheres. For each directional sphere image, only one (or a few when interpolation is done) sample is extracted, and the access pattern is somewhat arbitrary. Most compression techniques, which place some constraints on random access, are not appropriate for displaying the spherical light field because they often fail to decode an individual sample quickly.

Our compression method, based upon wavelets, produces as high compression ratios as the previous one used for the light field [15]. Our method is designed to be able to extract colors of individual samples, accessed in an arbitrary order, from the compressed data efficiently, using a modified significance map and an incremental decoding technique. It also provides a multi-resolution representation of the spherical light field that can be utilized in the future applications.

Wavelets are a mathematical tool for representing functions hierarchically, and they have had a great impact in several areas of computer graphics [8, 22]. They also provide powerful tools for multi-resolution representation of data and can be used for data reduction. Wavelets have been applied successfully to image compression, say [10, 21, 23] An image is compressed by applying a two dimensional wavelet transform to compute coefficients representing the image, and then disregarding the coefficients smaller in magnitude than a specified criterion.

Schröder et al. [19] discussed how to construct wavelets for functions defined on the sphere. Since the colors on the directional spheres are functions defined on the sphere too, their work is highly appropriate in this case. However, we choose to work on the reordered (i.e., *linearized*) image of the directional spheres. We aim at building a wavelet-based compression scheme which is appropriate not only for the spherical light field but also for the light field and

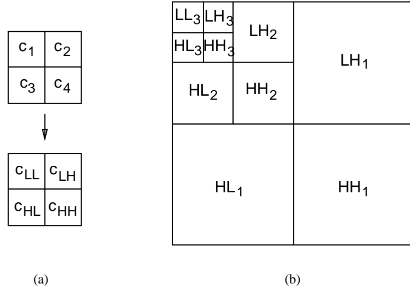


Figure 6. Wavelet Decomposition

lumigraph, based on the cubical representation. Furthermore, notice that each directional sphere is rather coarsely sampled, for example, 32×64 and 64×128 points when the discretization levels are 4 and 5, respectively. It would not be a good idea to compress data on each sphere handling them separately. Since the directional spheres corresponding to the neighboring points on the positional sphere contain much coherence, it would produce a better result to put several spherical data in one unit, and process them together, in which case, reordering into a 2D array gives a simple computational scheme.

In our implementation, we use the nonstandard Haar wavelet decomposition as in [3]. The Haar wavelet transformation is not one of the best wavelet filters, but it is computationally efficient, which is one of the most important factors in this application. Our experience tells that the Haar wavelets are good enough for compressing the spherical light field. Suppose an image is represented conceptually by a quadtree whose root corresponds to the whole image and whose descendants represent recursively subdivided regions. Decomposition is carried out by applying the Haar transform from the leaves all the way up to the root. The color values of the leaves can be thought to be the average colors of the corresponding regions represented by pixels. Then four average colors c_1, c_2, c_3, and c_4 in four children are recursively decomposed as one average color c_{LL} and three details c_{HL}, c_{LH}, and c_{HH} using the wavelet transform:

$$c_{LL} = \frac{c_1+c_2+c_3+c_4}{4},\ c_{HL} = \frac{c_1-c_2+c_3-c_4}{4},$$
$$c_{LH} = \frac{c_1+c_2-c_3-c_4}{4},\ \text{and}\ c_{HH} = \frac{c_1-c_2-c_3+c_4}{4}.$$

The four values arise from separable application of vertical and horizontal filters, where c_{LL} represents the average of pixel colors in the subregions, and c_{HL}, c_{LH}, and c_{HH} contain detail information on how the color varies in the four subregions. During decomposition, the four colors are replaced by the average color and three details (See Figure 6(a).), and then the average colors are repeatedly decomposed. The decomposition process converts the image into a wavelet image that can be stored in an array of the same size using a proper ordering of the coefficients.

As noted above, level 4 and level 5 discretization of directional spheres generates 64×32 and 128×64 images, respectively. In our compression scheme, we use as a unit image the image of size $2^7 \times 2^7 = 128 \times 128$ that amounts to 8 or 2 linearized directional spheres, discretized up to level 4 or 5, respectively. While the decomposition process produces a new wavelet image that has a reduced number of non-zero coefficients, it does not necessarily imply that the

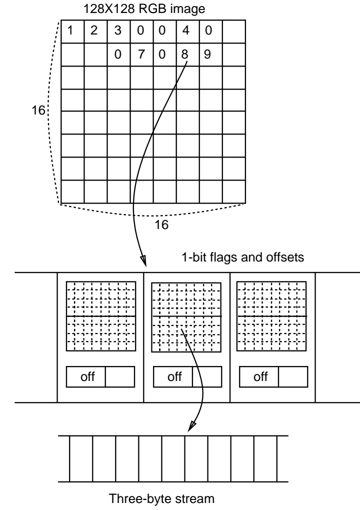
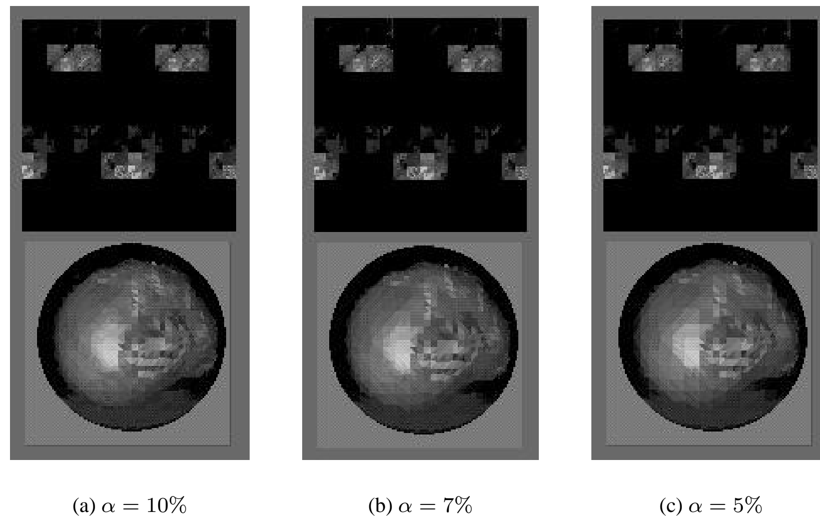


Figure 7. Wavelets Encoding Scheme

new image takes up less storage space. Consider an RGB image whose pixels are represented in three bytes. Each channel in the unsigned character type has a value ranging from 0 to 255. Since the repeated divisions during decomposition generate fractional numbers in each channel, the average colors and details cannot usually be stored in three bytes. Using floats (4 bytes), however, requires 12 bytes for color that may lead to an inefficient compression scheme. In fact, it is possible to exactly encode an image in integer arithmetic using slightly more bits per channel. For example, it is not hard to see that only 10 bits for each channel is sufficient when the so-called reversible S-transformation is employed to a 2D image [16].

In our method, we start from a 128×128 RGB image, in which three bytes are used for each pixel, and apply the wavelet transform three times to the image. Figure 6(b) shows a diagram of the resulting wavelet decomposed image in which the subscripts represent the levels of decomposition. The details in the blocks LH_1, HL_1, and HH_1 represent the finest scale wavelet coefficients and the first-level averages in LL_1 are again decomposed, and so on. Notice that the 16×16 block LL_3 contains the average colors, each corresponding to an 8×8 subregion of an image. During decomposition, we use enough precision, say four bytes per channel, to calculate the average and detail coefficients without round-off errors. Once the decomposition process is done, the coefficients are stored using one byte per channel, that is, three bytes per pixel. Particularly, the 16×16 averages are stored as unsigned integer, and the remaining details are stored in signed integer. Although the minimum precision required is 10 bits per channel when the S-transformation is employed, in which the floor operations must be carried out repeatedly, we choose the simpler Haar transformation for its computational simplicity. The information loss occurs when the coefficients are rounded off into three bytes. However, we find that the round-off errors in this stage do not cause much trouble in compressing the spherical light field. Now we have a 128×128 (partially) wavelet decomposed image whose coefficients are represented in three bytes. In the next stage, we delete (replace by zero) the vector-valued coefficients whose magnitude, measured in the L^2 norm, is smaller than a given threshold



(a) $\alpha = 10\%$ (b) $\alpha = 7\%$ (c) $\alpha = 5\%$

α	β	compr. ratio ρ
0.10	$\frac{100}{256}$	$\frac{3010}{49152}$ (6.1%)
0.07	$\frac{100}{256}$	$\frac{2532}{49152}$ (5.1%)
0.05	$\frac{86}{256}$	$\frac{2068}{49152}$ (4.2%)

Figure 8. Examples of Wavelet Compression

τ. Then only the non-zero coefficients are enumerated in a three-byte stream in the order described below.

As emphasized before, it is critical to be able to quickly reconstruct a color of an arbitrary pixel from a compressed image. Hence, there must be some mechanism that supports a quick random access. In our compression technique, the 128×128 image is partitioned into 16×16 subblocks, where each subblock represents 8×8 subimages (Figure 7). We first scan the subblocks in the left-to-right, up-to-down fashion, tagging with zero the subblocks whose coefficients are all zero, and with positive integers in the increasing order the subblocks that contains at least one non-zero coefficient. Since each tag can be represented using one byte, 256 bytes of storage is enough (Note that even when all the subblocks contains non-zero coefficients, 256 bytes are sufficient using some trick.). Each subblock with a non-zero tag is then scanned in the left-to-right, up-to-down fashion again, enumerating the non-zero coefficients in the three-byte stream. In addition, an auxiliary chunk of memory is allocated for this subblock that contains an 8×8 1-bit flag block and offset information. The one-bit flag block, requiring 8 bytes, contain the *significance map*, or the binary information as to whether the coefficients in the subblock are zero or not. The offset information *off*, represented in two bytes, contains the positions, in the three-byte stream, of the first non-zero coefficients in the ordering. A problem here is how to retrieve, efficiently, the value of a coefficient with index (i, j) in a decomposed 128×128 image whose non-zero coefficients are listed in the three-byte stream. To do this, we first check the tag of the subblock that contains the (i, j) element. If it is zero the coefficient is simply null. Otherwise we look at the auxiliary memory corresponding to the subblock that contains the 1-bit flags of the coefficients. Let (i', j') be the index of the coefficient (i, j) in the subblock. If the flag for the index (i', j') is 0, then the coefficient is zero. If not, we must carry out some computation to get the correct position or address of the (i, j) coefficient in the three-byte stream.

The position of the coefficient having the index (i', j') in the proper stream can be computed by adding its displacement value to *off*. When the flag is 1, the displacement is the number of coefficients that precede it in the enumeration whose flags are 1. To count the number efficiently, we use a precomputed indexing table $\mathsf{T}(*)$ with $2^{16} = 65536$ entries. Given a word made of two bytes, the table returns the number of bit 1 in the word. Hence, the correct number can be counted by accessing the table only a few times (Note that a proper number of zeros must be padded, in the word, from the position (i', j') position in the last access.).

We shall now analyze briefly, the costs that must be paid to access a coefficient in the compressed image. When the tag for the subblock that contains the coefficient is zero or its 1-bit flag is 0, that is, when the coefficient is zero, the cost is trivial. When its flag is 1, that is, when the coefficient is non-zero, a few table accesses, 2.5 on the average, and a few additions are necessary. The typical ratio of non-zero coefficients after wavelet compression we use in our implementation is less than 10 percent. This implies that accessing a coefficient in the compressed image involves little cost.

Once the non-zero coefficients have been enumerated in the three-byte stream, the coefficients are quantized. We view the data in the stream as a 24-bit true color image, and simply apply the vector quantizer that converts a 24-bit image into a 8-bit indices and a color table whose entries are three bytes long. In order to minimize the overheads caused by the color table, we build a color table for several, say four or eight, 128×128 images. Since the images corresponding to the neighboring vertices on the positional sphere are very

similar to each other, sharing one table does not harm the quality of vector quantization.

Extracting a color for a projection direction from the directional sphere is equivalent to reconstructing a color from a wavelet-encoded compressed image. The reconstruction process is the reverse of decomposition in which four average colors c_1, c_2, c_3, and c_4 are computed from one average color c_{LL} and three details c_{HL}, c_{LH}, and c_{HH} using the formulae :

$$\begin{aligned} c_1 &= c_{LL} + c_{HL} + c_{LH} + c_{HH}, \\ c_2 &= c_{LL} - c_{HL} + c_{LH} - c_{HH}, \\ c_3 &= c_{LL} + c_{HL} - c_{LH} - c_{HH}, \text{ and} \\ c_4 &= c_{LL} - c_{HL} - c_{LH} + c_{HH}. \end{aligned}$$

To extract the color of a specific pixel, that is, a specific direction, from the wavelet-encoded image, it is necessary to traverse the (conceptual) quadtree from the root down to the corresponding leaf, applying the above formula repeatedly. Since we have applied the wavelet transform three times, the reconstruction formulae are applied three times. To make the reconstruction computation as efficient as possible, it is carried out *incrementally*. Notice that as a mouse is moved to rotate objects in a scene, the projection directions of a point in the positional sphere change gradually. Hence, when we access a pixel in the image for the directional sphere, the chances are that it is near the pixel that has just been accessed. This implies that two adjacent reconstruction processes usually share a large portion of the paths from the root to the leaves in the quadtree. This observation suggests that the previous computation results may be reused and so, rather than discard the sets of four average colors for nodes in the previous path, we store them in a stack. (Note that only one copy of the previously accessed path is stored for each directional sphere.) When the next pixel is reconstructed, we compare the previous with the current paths to find the common path. Then the average colors of the common ancestors are reused without computing them again. This incremental computation enhances the performance of the reconstruction process.

Before turning to the next section, we report a quantitative analysis on the compression ratios. Firstly, a given 128×128 image takes up $2^7 \cdot 2^7 \cdot 3$ bytes. To store the tags for a compressed image, 256 bytes of memory is necessary (See Figure 7 again.). For a subblock that contains at least one non-zero coefficient, an auxiliary memory is allocated where 8 bytes (8×8 bits) are used for the 1-bit flags, and 2 bytes are used for the offset information. Let α be the ratio of the non-zero coefficients used after wavelet compression, that is, $\frac{\text{\# of coefficients in the three-byte stream}}{\text{\# of the whole coefficients used}}$, and β be the the rate of the subblocks with non-zero tags that contain at least one non-zero coefficient, that is, $\frac{\text{\# of non-null subblocks}}{\text{\# of the whole subblocks}}$. Then, the compressed image consumes $2^7 \cdot 2^7 \cdot \alpha + 10 \cdot 256 \cdot \beta + 256$, bytes and dividing it by $2^7 \cdot 2^7 \cdot 3$ bytes gives the compression rate $\rho = \frac{\alpha}{3} + \frac{5}{96}\beta + \frac{1}{192}$. This figure does not include the overheads for the color table: When the color table is shared by four and eight images, the additional cost are $\frac{256 \cdot 3}{2^7 \cdot 2^7 \cdot 3} = \frac{1}{64}$, and $\frac{1}{128}$, respectively.

Notice that the second and the third terms in ρ are the costs that must be paid to store the necessary significance maps of wavelet coefficients. In our scheme, this information allows a low-cost random access. It could be further compressed using the Zerotree or Horizon embedded coding techniques, but that only places constraints on random access to the compressed data. Figure 8 illustrates three examples of compression from the directional sphere image shown in Figure 5.

The compression ratios we usually achieve with this encoding scheme for the directional sphere data range from 10:1 to 30:1, depending on the coherence in data. To design a compression technique using wavelets, we have compromised between a good compression ratio and fast random access ability. Our compression technique could be useful for other applications that can benefit from the multiresolution representations.

Figure 9. Ray Casting of the Human Head with Skin

4. Experimental Results

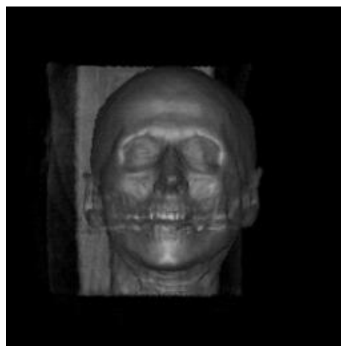
The complete image-based rendering system has been implemented on an SGI Indigo 2 workstation with a 200MHz R4400 CPU, 256 Mbytes of main memory and a High IMPACT graphics board. The performance results for three spherical light fields with the different parameters are summarized in Table 1. The test spherical light fields were created using our volume ray casting software from the "UNC head" data set which is a $256 \times 256 \times 225$ CT scan of a human head. We created four spherical light fields for two classifications and two discretization levels of the directional spheres. The raw spherical light fields we have generated take about 384 Mbytes to 1.5 Gbytes of storage. The data size is proportional to the number of vertices in the positional sphere, and to the number of triangles in the directional spheres.

Figure 9 shows a 256×256 image generated with a parallel projection of the CT data classified with an opaque skull and semitransparent skin. It was rendered directly from the CT data by our volume rendering software for comparison purposes. We used a ray casting algorithm which is optimized using octrees and early ray termination [14]. In the implementation, four bytes (one for density, two for normal directions, and one for normal's magnitude) were allocated per voxel for fast classification and lighting computation, hence it takes about 57 Mbytes of memory for the resolution $256 \times 256 \times 225$. Each rendering usually took longer than a minute.

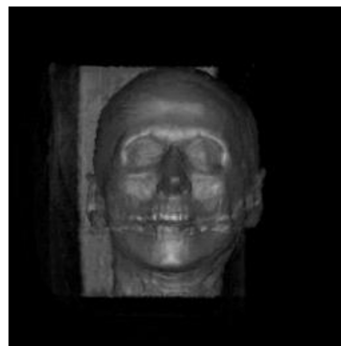
The four images in the following two figures were extracted from two spherical light fields with the level 4 resolution for the directional sphere. Two spherical light fields of level 4 discretization were constructed. For the first one, the positional sphere was created by recursively subdividing

Positional Sphere			Dis. level for dir. Sphere	Size of raw data (MBytes)	Ratio of Non-zero wavelet coefficients used	Size of compressed data (MBytes)	Compression Ratio
base	# of vertices	# of triangles					
8	65,538	131,072	4	384.0	15 %	35.2	10.9 : 1
					10 %	28.6	13.5 : 1
20	40,962	81,920	4	233.5	15 %	22.0	10.6 : 1
					10 %	17.8	13.1 : 1
16	32,770	65,536	5	768.1	10 %	49.9	15.4 : 1
					5 %	34.3	22.4 : 1

Table 1. Result of Data Compression

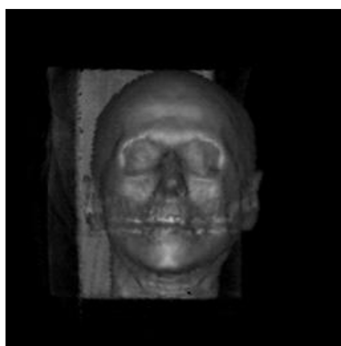


(a) 15%, 35Mbytes

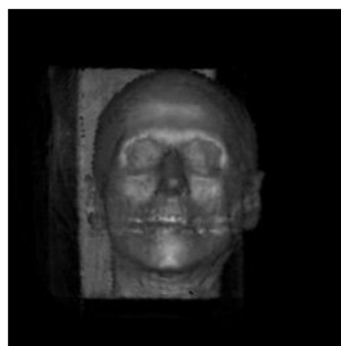


(b) 10%, 29Mbytes

Figure 10. Level 4 Renderings (an Octahedral Base)



(a) 15%, 22Mbytes



(b) 10%, 18Mbytes

Figure 11. Level 4 Renderings (an Icosahedral Base)

an octahedral base up to level 7 where the total size of the data amounts to 384.0 Mbytes. For Figure 10(a) and (b), the spherical light fields were compressed by deleting 85 and 90 percents of wavelet-encoded coefficients in the decomposed wavelet images, where the resulting compressed data take 35 and 29 Mbytes of storage, respectively. These data could be further compressed using the entropy coder such as gzip, in which case the data sizes become 27 and 21 Mbytes. (These figures are not important when we are interested in the run-time memory requirement.) The positional sphere of the second level 4 data was generated by level 6 subdivision of an icosahedral base, where the total size amounts to 233.5 Mbytes. The images in Figure 11(a) and (b) were generated from the spherical light fields that were compressed by deleting 85 and 90 percents of wavelet-encoded coefficients in the decomposed wavelet images, where the resulting compressed data take 22 and 18 Mbytes of storage, respectively.

When 15 % of coefficients are used, the rendering produced images whose quality is competitive with that of ray casting. We observe that the rendered images still maintain a good quality when 10 % of the coefficients are used.

To measure the running time for the level 4 spherical light fields, the head was rotated 360 times at one degree angle increments. For the first data, it took about 0.44 second per frame on the average. To render the second data which has fewer vertices on the positional sphere, it took about 0.29 second per frame on the average. Notice that the rendering time is proportional to the number of vertices on the positional spheres. The image-based rendering task is broken down into two major computations: 0.07 to 0.11 second is consumed per frame for polygonal rendering (viewing and shading) of the tessellated positional sphere. This part is accelerated by the graphics hardware, and the running time is almost independent of output image size. The remaining portion of rendering time is spent mostly in accessing 65538 and 40962 (for each data, respectively) compressed directional spheres to associate colors with the vertices of the positional sphere. It is run in software, and hence relies on the speed of CPU. Our preliminary implementation on a PC with a 200MHz Pentium Pro CPU, 128 Mbytes of main memory and an Intergraph Intense 3D graphics board (without texture memory), produces a little faster timing performance.

The timings per frame are found somewhat irregular, which is due to the way the wavelet-compressed directional spheres are accessed. To get a color from a directional sphere, its conceptual quadtree is traversed downward corresponding to a given projection direction, applying the reconstruction formulae to each node on the path. The effect of the incremental computation stands out when a large portion of the paths are shared by two adjacent accesses. When the object is rotated gradually, the pixels corresponding to two adjacent accesses are very close to each other in the images. In probability, two close pixels have most ancestors in common, but that is rather irregular depending on their locations. (Two adjacent pixels can share only the root in quadtrees.) Notice that the first rendering takes more computations because the whole stacks of the directional spheres must be constructed. Then, the decreased timings for the following renderings show how the incremental traversal technique works.

In Figure 12, three images from the spherical light field with the level 5 discretization for the directional sphere, are shown. Two data sets for (a) and (b) were compressed so that 10 and 5 percent, respectively, of coefficients in the wavelet encodings were used. The resulting compressed spherical light fields took 50 Mbytes and 34 Mbytes, respectively. The image in (c) was produced with a perspective projection. For the level 5 data, the object was also rotated 360 times by one degree, and it took about 0.26 second per frame on the average. It took roughly twice as long for perspective because the projection directions had to be computed for each vertex. As mentioned earlier, when a parallel projection is used, the level 5 does not necessarily produce output images of higher quality. The image quality is more affected by how many coefficients are deleted in compression. The higher compression ratios are required for the level 5 data due to their huge size, and they degrade image quality more as a result. The projection directions on the directional spheres are more densely sampled when the level is 5. This results in a smoother transition in an animation movie, however, we observe that the level 4 is good enough in most cases.

The last figure (Figure 13) shows how naturally our object-space image-based rendering algorithm intermingles with traditional polygonal rendering. (The jagged stuff behind the head is a material in the CT data, classified with bone and skin.) The head was rendered from a spherical light field, and the rest was rendered from polygonal models. In the context of polygonal rendering, the tessellated positional sphere can be regarded as a special graphics primitive that is made up of triangles and color information at the vertices. Adding this new graphics primitive will extend the conventional polygonal rendering into a simple-to-implement hybrid rendering in which complex regions are rendered from spherical light fields, while less complex regions are rendered from simple polygonal models.

5. Conclusions

In this paper, we have described an image-based rendering framework, based on a new parameterization of the flow of light in space. We showed that the spherical light field is an effective representation that allows efficient computations, and is easily added in the graphics pipeline based on polygonal rendering. In addition, we introduced an encoding scheme based on wavelets for compression of the huge data resulting from sampling of the spherical light field.

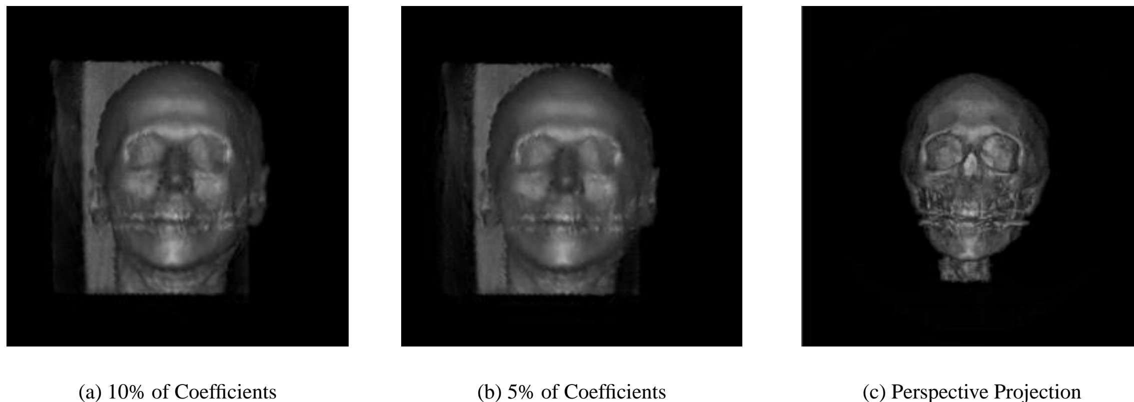
We are currently investigating the relationships between the number of triangles and vertices in the tessellated positional spheres, and the image quality. The data size is proportional to the number of vertices, hence reducing the number without degrading image quality will enhance the timing and space performances. Since, in our implementation of wavelet-compression, one unit contains two or eight directional spheres with the same (θ_p, ϕ_p) coordinates, we could say our encoding is $(2+\alpha)$-dimensional. We are now compressing the 4D spherical light fields using 3D and 4D wavelets hoping to get higher compression ratios without harming decoding speeds.

Acknowledgments

We would like to thank an anonymous reviewer for the useful comments. This work was supported in part by the Ministry of Science and Technology of Korea through the STEP2000 project.

References

[1] E. H. Adelson and J. R. Bergen. The plenoptic function and the elements of early vision. In M. Landy and J. A.



(a) 10% of Coefficients (b) 5% of Coefficients (c) Perspective Projection

Figure 12. Level 5 Renderings - a Base that has 16 polygons

Movshon, editors, *Computational Models of Visual Processing*, chapter 1. The MIT Press, Cambridge, Mass., 1991.

[2] T. Beier and S. Neely. Feature-based image metamorphosis. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 35–42, 1992.

[3] D. Berman, J. Bartell, and D. Salesin. Multiresolution painting and compositing. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 85–90, 1994.

[4] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *CACM*, 19(10):542–547, October 1976.

[5] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, Utah, December 1974.

[6] S. E. Chen. Quicktime VR - an image-based approach to virtual environment navigation. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 29–38, 1995.

[7] S. E. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 279–288, 1993.

[8] C. K. Chui. *An Introduction to Wavelets*. Academic Press Inc., 1992.

[9] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 11–20, 1996.

[10] R. DeVore, B. Jawerth, and B. Lucier. Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, 38(2):719–746, March 1992.

[11] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of Visualization '90*, pages 176–186, San Francisco, 1990.

[12] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, 1996.

[13] N. Greene. Environment mapping and other applications of world projections. *IEEE CG & A*, 6(11):21–29, November 1986.

[14] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[15] M. Levoy and P. Hanrahan. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, 1996.

[16] P. Lux. A novel set of closed orthogonal functions picture coding. *Arch. Elek. Übertragung*, 31:267–274, 1977.

[17] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 39–46, 1995.

[18] B. O'Neill. *Elementary Differential Geometry*. Academic Press, 1966.

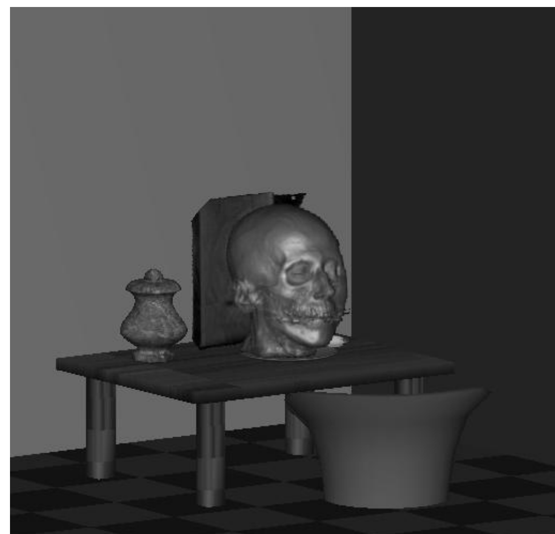


Figure 13. Image-Based Rendering Intermingled with Polygonal Rendering

[19] P. Schröder and W. Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 161–172, 1995.

[20] S. M. Seitz and C. R. Dyer. View morphing. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 21–30, 1996.

[21] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[22] E. Stollnitz, T. DeRose, and D. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers, Inc., 1996.

[23] A. Zandi, J. D. Allen, E. L. Schwartz, and M. Boliek. CREW: Compression with reversible embedded wavelets. In *Proc. of Data Compression Conference*, pages 212–221, Snowbird, Utah, March 1995. IEEE.